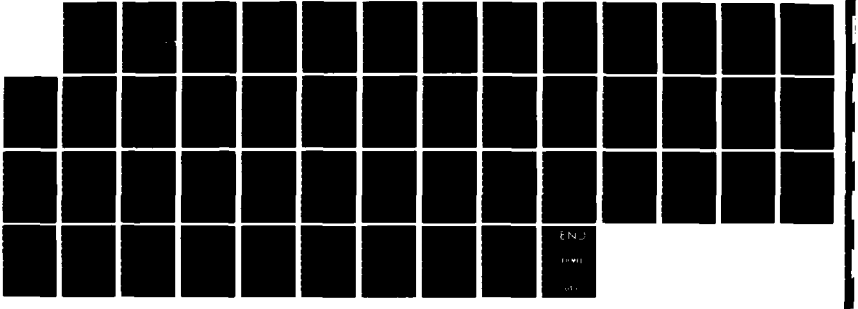
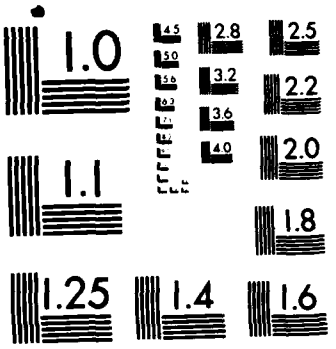
AD-A162 221 NUMERICAL ALGORITHMS & PARALLEL TASKING(U) 1/1
MASSACHUSETTS INST OF TECH CAMBRIDGE V KELMA 12 SEP 85
AFOSR-TR-85-0988 AFOSR-82-0210

UNCLASSIFIED F/G 12/1 NL





Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

AD-A162 221

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS:
2a. SECURITY CLASSIFICATION AUTHORITY: ---	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for Public Release; Distribution Unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S):	5. MONITORING ORGANIZATION REPORT NUMBER(S): AFOSR-TR- 85-0988
6a. NAME OF PERFORMING ORGANIZATION: Massachusetts Inst. of Tech. / 6b. OFFICE SYMBOL (If applicable):	7a. NAME OF MONITORING ORGANIZATION: AFOSR
6c. ADDRESS (City, State and ZIP Code): Cambridge, MA 02139	7b. ADDRESS (City, State and ZIP Code): Bldg. 410 Bolling AFB, D.C. 20332-6448
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: AFOSR / 8b. OFFICE SYMBOL (If applicable): NM	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: AFOSR-82-0210
c. ADDRESS (City, State and ZIP Code): Bldg. 410 Bolling AFB, D,C, 20332-6448	10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.
61102F	2304	A3	

11. TITLE (Include Security Classification): Numerical Algorithms & Parallel Tasking (U)

12. PERSONAL AUTHOR(S): Virginia Kelma

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
Annual	FROM 12 Oct 84 TO 15 Sep 85	12 September 1985	48

16. SUPPLEMENTARY NOTATION

DTIC ELECTE DEC 9 1985 S A

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	parallel tasking, control-synchronization

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

COPY

The first version of the parallel tasking has been completed, installed on three of the project's workstations, and tested using several different control-synchronization structures. Two papers entitled "Fast Toeplitz Orthogonalization Using Inner Products" and "Fast Singular Value Decompositions of Some Structures Matrices" describe algorithms being adapted for aprallel implementation on the project's concurrent system.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☐	Unclassified

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE NUMBER (Include Area Code)	22c. OFFICE SYMBOL
John P. Thomas, Jr., Capt., USAF	(202) 767-5026	NM

DD FORM 1473, 83 APR — EDITION OF 1 JAN 73 IS OBSOLETE.

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

PROGRESS REPORT

Numerical Algorithms and Parallel Tasking
AFOSR-82-0210

Principal Investigator, Virginia C. Klema
Massachusetts Institute of Technology

BACKGROUND

The long term goals of this research activity are derived from concurrent computing with emphasis on numerical algorithms that support a variety of scientific applications. Among the applications of immediate interest are signal processing, high resolution spectrum estimation, and computationally intensive statistical methods such as the bootstrap. The shorter term focus has been the refinement and enhancement of the concurrent computing environment itself and the numerical algorithms that form the foundation for the applications.

The computer systems that support this research use hardware manufactured by Intel -- with the 8086 cpu and the 8087 floating point processor at the heart of each node. The nodes are contained in workstations (with up to seven Workers and a Manager in a workstation) that can operate independently or can be linked together during execution. Worker structure and memory are configurable to support experiments conducted in the course of this research.

PERSONNEL

During the period covered by this report, May 15, 1984 to July 15, 1985, the following senior personnel have been supported under this contract: Virginia Klema, principal investigator (3.5 months), Elizabeth Ducot (2.25 months), and George Cybenko (1.1 months). In addition, research staff member Richard Kefs has spent 4.5 months on the project; while 1.5 months of graduate student support has been provided.

STATUS

Version 1.0 of the Tasker (a comprehensive concurrent computing environment) has been completed, installed on three of the project's Concurrent Computing Workstations, and tested using several different control-synchronization structures. By deliberate intent, the segmenting of modules for computation and the degree of concurrency requested are the responsibility of the user. The software Tasker provides primities to support assignment of concurrency within the algorithm or the

application. On-line help utilities have been created that provide support for the use of the Tasker through all stages of software development and testing. The primitives that permit the user to define his environment, locate code and data on the worker processors, pass data and messages through the system, and monitor execution of the various portions of his algorithm are described in a project working paper, WP4, entitled "Application Interface to the Concurrent Environment".

A principal challenge for our research is the melding of numerical analysis, algorithmic design for concurrency, specific applications, and analysis of concurrency. A favorable prospect for this effort is the graceful migration of the basic computational modules and primitives for the software tasker to large scale MIMD machines.

WP4: AFOSR-82-0210
Rev: July 1, 1985

APPLICATION INTERFACE To The CONCURRENT ENVIRONMENT

Elizabeth R. Ducot

The purpose of this note is twofold. The first is to present the mechanisms by which a user activates and describes his concurrent computing environment (See Section I, System Activation). The second is to define the system routines and functions by which he controls the real-time execution of his application programs within that environment (See Section II, Real-Time Execution).

In both cases, the primitives introduced are presented as a "first effort" -- planning for a more sophisticated user environment that includes tools for debugging and monitoring is already underway. The concurrent environment, called the **TASKER**, consists of two separate operating systems: 1) the **MANAGER**, the multi-tasking operating system running on the master processor, and 2) the **WORKER**, the smaller self-contained system running on each of the worker processors. The primitives within the **TASKER** (whether for definition of the environment or execution) are activated via statements inserted in the user's code of the form:

CALL **function** (parameters...).

Throughout the discussion that follows, samples of these statements appear as part of the general explanation. However, specific instructions on how to use each of the commands and how to define and interpret each of the parameters in the calling sequences have been deferred to Appendix A.

A complete application consists of three distinct components: 1) the initialization software, 2) the control program which coordinates the activities of the workers and may interact directly with the user, and 3) the portions of the application designed to execute in parallel solely on the workers. The initialization software must execute first to activate the system as described below.

I. SYSTEM ACTIVATION

1.1 System Initialization

The user initializes the concurrent computing environment programatically. This initialization program can be written in any language and is invoked from the user's terminal. It executes on the master processor and is linked to the **MANAGER**[1] in

order to resolve system references. The first statement in the initialization sequence is:

CALL **init_concur** (1)

This function initializes the worker processor boards and passes parameters throughout the system that ensure consistent behavior of the floating-point arithmetic support environment. At the same time, this call establishes that all workers are active and that the individual **WORKER** operating systems can communicate with the **MANAGER**. Upon completion of this command, a summary is presented to the user indicating which physical processors are currently available. In the event that this does not match the user's perception of the physical system he has programmed, he may elect to abort the initialization, rework his code or request a reconfiguration of the hardware!

1.2 System Interconnect (OPTIONAL)

The second stage in the initialization is an optional one in which the user tailors the communication paths in the system to his application. Before he can customize this interconnect, he should have a working knowledge of how the communication paths are established when the default interconnect structure is used.

Communication between worker processors is accomplished via messages that are "posted" to communication ports. Each worker owns a maximum of n input and n output ports that are used to create virtual circuits. If the user does nothing explicit, the default is an "any to any" interconnect structure in which the physical unit numbers of the other n-1 workers (those displayed during initialization) plus the manager are mapped into the 2n logical port identifiers. For a given worker "i", output ports (corresponding to logical units for writing) take on the value of the physical unit number (1 to n) of potential receivers of messages from "i", while input ports are assigned physical unit numbers (1 to n) of processors that may be sending messages to processor "i". Unit 0 is reserved in both cases for the manager itself.

The most obvious reason for overriding the default is the case in which the user wishes to place the same code, utilizing the same logical unit numbers for reading and writing, on each processor. Physical units for reading and writing will be necessarily different, depending on which physical processor is actually executing the particular copy of the code. In order to assign these processor specific logical unit numbers, the following call is made for each processor requiring an override.

CALL **send_structure** (proc_id, in_list, out_list, status) (2)

[1] The linkage to the **TASKER** library for the manager is defined in Appendix B.

A secondary consideration in deciding whether or not to override the communication structure may be one of efficiency. Some improvement in performance is expected to result from limiting the "any to any" interconnect structure to accommodate only those message ports required by the application. However, this consideration should be ignored at the outset in the interest of establishing an initial test case.

1.3 Movement of Code

Once the communication structure has been established the user must then indicate the placement of his code across the processors at his disposal. A subroutine call is required to specify the disposition of each unique code segment to be used in the application. The same code may be sent either to one or broadcast to many processors with a single function call. The form of this call is the following:

```
CALL send_code (num_processors, list_of_processors,
     file_name, execution_mode, list_of_code_ids, status)
                                                        (3)
```

Code segments are prepared in advance, stored on the disk, and moved from the file indicated in the calling sequence to the desired processors. The user may develop as many or as few of these distinct code segments as he wishes. Depending upon the execution_mode, execution may begin as soon as the code is received at its destination or may be suspended awaiting an explicit "**start**" command.

II. REAL-TIME EXECUTION

2.1 Structure of the Application

The user has the option of placing his control program on a specific worker processor or running it directly on the manager. For the time being, the user is encouraged to run his control program on the manager. There are a number of reasons for this; the following are the most significant. First, since only the manager has direct access to the terminal and the disk, debugging and data handling are made easier for the user who includes the manager within his real-time execution environment. Secondly, the processing power of the manager is currently underutilized, leaving ample resources to support control and synchronization type statements.

This will not always be the case. As the development of the **MANAGER** proceeds, it is expected that a more sophisticated debugging environment for both manager and worker will be specified. A number of new monitoring functions will be defined, with the suite of functions and commands available to the user and/or system continuing to grow. In the future, therefore, the load on the manager may be such that the execution of the control

program on the manager processor itself would interfere with **MANAGER**'s ability to respond to requests from the other processors in the system.

If the control program is designated for the master processor, it is linked along with the initialization software to the **MANAGER** portion of the **TASKER**. Worker software, on the other hand, whether for control or computation is linked to the more limited **WORKER** environment. The **WORKER** supports real_time execution statements only; assuming that the environment has been fully initialized by the manager. In addition, I/O capabilities on the workers are limited and are provided only by the **TASKER** primitives themselves. The particular restrictions on the code segments depend on the implementation language and are described elsewhere.[2] At the end of each module, regardless of location is a statement of the following form:

CALL **complete** (4)

which does not return control to the user.

2.2 Initialization

Real-time execution is initiated with the following function call:

CALL **execute_concur** (control_mode) (5)

A value of 0 for the control_mode indicates that a control program has been set up to run on the manager. The subtroutine call to the control program or inline code will be executed immediately after returning from the call to execute_concur.

A value of 1 for the control_mode indicates that the control program has already been posted to a worker via a send_code command. In this case, the **MANAGER** takes over all the resources of the master processor and does not return control to the user at the terminal.

From this point on, the progress of the execution is governed by the interaction among the programs running on the various processors as well as the control program.

2.2 Movement of Data

Both synchronous and asynchronous communication primitives are provided. Asynchronous calls are extremely fast and merely establish the communication request. However, the data buffers

[2] See Appendix B for a discussion of the use of either PL/M or FORTRAN. Requirements for using C will be added at a later date.

associated with asynchronous calls, whether for outgoing or incoming data, cannot be immediately reused. Synchronous calls, on the other hand, block the processor until 1) transmission of the data is complete in the case of outgoing messages, or 2) the message has been completely received in the case of incoming data. Data buffers associated with synchronous calls, therefore, are available immediately on return to the user. Blocking calls should be used with extreme caution since control does not return to the user until they have been completed satisfactorily. A more efficient approach, from the point of view of utilizing the computing power of the processor, is to mix asynchronous calls with interrogation of the transaction status (see **idstatus**) and the computational statements that comprise the rest of the algorithm.

During a send operation, data may be either broadcast or point-to-point; the syntax of the send command is the same:

CALL **send_data** (number_of_ports, list_of_port_ids,
data_buffer, number_of_entries, precision_code,
list_of_message_ids, status) (6 & 7)

The preceding routine generates an asynchronous call. The synchronous command to send data has the same syntax, however, the subroutine name is **s$send_data**.

The command to receive data is similar to the send command, except that only point-to-point data movement is supported. The asynchronous form of this function is:

CALL **receive_data** (port_id, data_buffer, number_of_entries,
precision_mode, message_id, status)) (8 & 9)

Again, the synchronous form of the command, activated under the name **s$receive_data**, uses the same syntax.

2.3 Utilities

The most important of the utilities is the function for determining the system status. This is essential in the case of an asynchronous call, since only when the transaction_status of a specific transaction is complete can the buffer associated with that transaction be used again. The form of this call is :

transaction_status = **idstatus** (transaction_id) (10)

where transaction_id refers either to a message_id or a code_id.

The remaining utilities are optional, since coordination may be achieved entirely through mixtures of **send_data, receive_data and status** calls. However the **start** and **collect** commands may be very useful to the user who desires to force synchronization or control specific timing.

CALL **start** (num_ids, list_of_code_ids, fork_status) (11)

broadcasts an initiation message to all processors running code segments on the list. This extremely short message results in nearly simultaneous initiation on each processor. It should be noted that unless worker code is already executing (e.g. has been "started" by the *send_code command*), a **start** command must be issued before any message traffic can be processed by the worker.

CALL **collect** (num_ids, list_of_ids, join_status) (12)

monitors either the completion messages sent by the code segments on each worker or the status of specific transactions in the list.

CALL **byte_copy** (input_vector, output_vector,
number_of_entries, precision_mode, move_status) (13)

This utility permits copy of floating point elements without using the 8087, particularly important if the movement of NAN's is desired.

The final function available to the user in the concurrent environment allows him to determine the physical unit number of the processor on which he is executing from inside the program. The form of the call is:

CALL **whoami**(proc_id) (14)

A P P E N D I X A

The components of the concurrent computing environment accessible to the user are described below. The discussion that follows presumes that the application language is FORTRAN -- the only one of the Intel languages that provides a set of language extensions for taking full advantage of the numeric co-processor in the system. Within this framework, the following conventions are used in defining the variables:

1) an INTEGER is a two byte signed quantity (with a value in the range -32,768 to +32,767) and is defined within the application as INTEGER*2.

2) a BYTE corresponds to the declaration INTEGER*1.

3) An "untyped" data buffer refers to an array of any legitimate type. It is important to note that all calls to tasker routines are made by reference; the address of the data buffer is all that is known by the **TASKER**. No type or bounds checking is done when processing a buffer. Thus, the responsibility for allocating sufficient buffer space [(num_entries)*(prec_code) bytes] to accommodate receive and copy requests remains the responsibility of the user.

All concurrent primitives, presented below in alphabetical order, are accessed via subroutine calls from the FORTRAN application program. The single exception to this procedure is the status check function (**idstatus**), an INTEGER function which may be used as part of a logical IF statement. Five of the commands listed, marked SA (System Activation) only, are used during the portion of the application dedicated to initializing the concurrent environment. Thus these commands are available only to software linked to the **MANAGER** operating system and running on the master processor. All other primitives are used identically whether executing on the master or worker processors.

One further comment is appropriate regarding the output of these routines. In many cases an INTEGER return code has been defined (appearing in the calling sequence as a parameter whose name ends with the characters "_status"). In general, these parameters are included in the specifications of the **TASKER** primitives for completeness only -- to allow for future development of the software. This version of the **TASKER** (Version 1.0) treats abnormal behavior of its commands as "unrecoverable errors" and therefore returns control to the user only when no exceptional conditions have been encountered. When sending code to worker processors, however, this parameter is significant and should be examined. (See **send_code**).

CALL **byte_copy** (in_vector, out_vector, num_entries,

prec_mode, move_copy)

INPUT PARAMETERS:

in_vector	An untyped data buffer containing the data to be copied.
out_vector	An untyped destination buffer.
num_entries	An INTEGER giving the number of data items in the data buffer. NOTE: The number of bytes that will be copied is: (num_entries) * (prec_code).
prec_code	A BYTE indicating the number of bytes required for an entry in the data buffer. Thus prec_code takes on the value of: 1 = byte, integer*1, or character 2 = 2 byte integer or word 4 = long integer or short precision floating point number 8 = long precision floating point number 10 = extended precision or TEMPREAL

OUTPUT PARAMETERS:

move_copy	An INTEGER return code indicating the number of bytes moved successfully.

CALL **collect** (num_ids, trans_ids, join_status)

INPUT PARAMETERS:

num_ids	An INTEGER indicating the number of entries in the list of trans_ids to be processed.
trans_ids	An INTEGER vector containing the transaction identifiers associated with either: 1) code executing on each of the designated worker processors. The identifiers were obtained when the code was sent to a worker via a send_code command. 2) a list of messages to be "collected"

OUTPUT PARAMETERS:

join_status An INTEGER return code defined to reflect the following conditions:

0 = no exceptional conditions--execution of all code indicated in the list of code_ids or transaction of all messages listed is complete.

NOTE: THIS ROUTINE IS A SYNCHRONIZATION POINT. CONTROL WILL NOT RETURN TO THE USER UNTIL COMPLETION MESSAGES FOR ALL TRANSACTIONS ON THE LIST HAVE BEEN DETECTED OR A FATAL ERROR HAS BEEN RECEIVED.

CALL **complete**

INPUT PARAMETERS: -- NONE --

OUTPUT PARAMETERS: -- NONE --

This call must appear as the last statement in each module running on the worker processor, as well as the last statement in the manager program. Control never returns to the user after issuing this statement.

CALL **execute_concur** (control_mode) (SA only)

INPUT PARAMETERS:

control_mode A BYTE variable indicating the location of the user's control program.

0 = control program is to run on the manager. In this case, user's control statements follow the return from this subroutine.

1 = control program has already been placed on its appropriate worker. User statements encountered after this point will be ignored and **TASKER** will be in complete control of the processor.

OUTPUT PARAMETERS: -- NONE --

---

trans_status = **idstatus** (trans_id)

INPUT PARAMETERS:

trans_id — An INTEGER referring either to a message identifier or a code identifier.

OUTPUT PARAMETERS:

trans_status — An INTEGER defined as follows:

0 = original request to initiate the transaction has been logged.

1 = transaction queued

2 = request for data sent (applicable to receive requests only)

3 = transaction partially completed

4 = transaction complete.

CALL init_concur (SA only)

INPUT PARAMETERS: -- NONE --

OUTPUT PARAMETERS: -- NONE --

CALL notify_in (routine)

INPUT PARAMETERS:

routine — A BYTE designating a user defined routine number (0-99) used to trace logic in the event of a fatal error.

OUTPUT PARAMETERS: -- NONE--

CALL notify_out

INPUT PARAMETERS: -- NONE --

OUTPUT PARAMETERS: -- NONE --

NOTE: Calls to notify_in and notify_out must be used in pairs in order that the traceback information, provided in the event of a fatal error, make sense. Good practice would dictate that notify_in become the first statement executed on entry to a user's main program or subroutine; notify_out the last prior to a RETURN or an END. Any selection of routine numbers in the legal range is acceptable, however a call to notify_in with the routine number=0 has the additional effect of insuring that all 8087 exceptions are unmasked.

CALL **receive_data** (port_id, data_buffer, num_entries, prec_mode, message_id, input_status)

INPUT PARAMETERS:

port_id — A BYTE containing the logical unit number associated with the intended receiver.

data_buffer — An untyped data buffer reserved for the requested message.

num_entries — An INTEGER giving the number of data items expected to be placed in the data buffer. NOTE: This may or may not correspond to the number of bytes, depending on the size of each entry.

prec_code — A BYTE indicating the number of bytes required for an entry in the data buffer. Thus prec_code takes on the value of:

1 = byte or character
2 = 2 byte integer or word
4 = long integer or short precision floating point number
8 = long precision floating point number
10 = extended precision or TEMPREAL

OUTPUT PARAMETERS:

message_id — An INTEGER containing the transaction identifier for this request for data.

input_status — An INTEGER return code defined to reflect the following conditions:

0 = no exceptional conditions

CALL **send_code** (num_procs, processors, file_name,
exec_mode, code_ids, send_status)

(SA only)

INPUT PARAMETERS:

num_procs — A BYTE variable indicating how many workers are to receive the code segment referenced in this function call

processors — A BYTE vector giving the "num_procs" physical unit numbers of those workers

file_name — A string of CHARACTERS indicating the complete path name of the code file on the disk. (See Appendix B for instructions on file preparation.) The terminator for the string is the character "!".

exec_mode — A BYTE controlling initiation of the code segment on the worker:

0 = begin execution immediately

1 = defer execution until worker receives an explicit **start** command.

OUTPUT PARAMETERS:

code_ids — An INTEGER vector containing the transaction identifiers associated with each of the designated worker processors. Moreover, code_id(i) contains the identifier corresponding to code running on processor(i) in the input list.

code_status — An INTEGER return code defined to reflect the following conditions:

0 = no exceptional conditions

1 = at least one worker in the list is temporarily unavailable. Code has been sent to all processors in the list processor(i) for which legitimate code_ids(i) have been assigned. A code_id of -1 indicates that code was not transmitted to that worker.

CALL **send_data** (num_ports, port_ids, data_buffer,
num_entries, prec_code, message_ids, out_status)

INPUT PARAMETERS:

num_ports — A BYTE value indicating the number of receivers of a broadcast message. If num_ports = 1, the message is point-to-point.

port_ids — A BYTE vector containing a list of logical unit numbers associated with the intended receivers.

data_buffer — A POINTER to an untyped data buffer containing the contents of the message to be sent.

num_entries — An INTEGER giving the number of data items in the data buffer. NOTE: This may or may not correspond to the number of bytes, depending on the size of each entry.

prec_code — A BYTE indicating the number of bytes required for an entry in the data buffer. Thus prec_code takes on the value of:

1 = byte or character
2 = 2 byte integer or word
4 = long integer or short precision floating point number
8 = long precision floating point number
10 = extended precision or TEMPREAL

OUTPUT PARAMETERS:

message_ids — An array of INTEGERS of sufficient length to hold one message identifier for each entry in the list of receiving ports.

out_status — An INTEGER return code defined to reflect the following conditions:

0 = no exceptional conditions

CALL **send_structure** (proc_id, input_list, output_list, com_status) (SA only)

INPUT PARAMETERS:

proc_id — A BYTE specifying the physical unit number of the processor receiving the modified I/O lists.

input_list — A BYTE vector defining the mapping between

physical unit numbers of the other workers in the system and logical units for input as seen by worker "proc_id". The first entry in the list becomes logical unit 1, etc.; a value of -1 stored in an INTEGER*1 variable (or FFh) indicates that the logical unit will not be used. The manager is automatically assigned to logical unit (and physical unit) 0. If one wishes to assign the manager to an alternate logical unit number, he must use the physical unit number -2 (or FEh).

For the current hardware configuration, the maximum length of this vector is 7. However, all 7 entries need not be present; a value of 0 supplied at any point will terminate the search for valid logical unit numbers.

EXAMPLE: Suppose one wishes logical unit 1 to correspond to input from processor 5, logical unit 3 to input from worker 6, and logical unit 4 to input from the manager. In addition supose that no other input units are required by this application. The input_list would be defined as follows:

```
input_list = 5, -1, 6, -2, 0
```

output_list — A BYTE vector of the same form as "input_list" to define the logical units for output.

OUTPUT PARAMETERS:

com_status — An INTEGER return code defined to reflect the following conditions:

0 = no exceptional conditions

CALL set_breakpoint (loc_count)

INPUT PARAMETERS:

loc_count — A BYTE defined by the user to mark a specific location in the user's code. Calls to set_breakpoint may be placed anywhere between calls to notify_in and notify_out. The most recent value of loc_count, along with the routine number given by the currently active **notify_in** command, will be returned in the event of a fatal error.

OUTPUT PARAMETERS: -- NONE --

CALL **s$receive_data** (port_id, data_buffer, num_entries,
prec_mode, message_id, input_status)

INPUT PARAMETERS:

port_id — A BYTE containing logical unit number associated with the intended receiver.

data_buffer — An untyped data buffer set aside to receive the message.

num_entries — An INTEGER giving the number of data items in the data buffer. NOTE: This may or may not correspond to the number of bytes, depending on the size of each entry.

prec_code — A BYTE indicating the number of bytes required for an entry in the data buffer. Thus prec_code takes on the value of:

1 = byte or character
2 = 2 byte integer or word
4 = long integer or short precision floating point number
8 = long precision floating point number
10 = extended precision or TEMPREAL

OUTPUT PARAMETERS:

message_id — An INTEGER containing the transaction identifier for this request for data.

input_status — An INTEGER return code defined to reflect the following conditions:

0 = no exceptional conditions

CALL **s$send_data** (num_ports, port_ids, data_buffer,
num_entries, prec_code, message_ids, out_status)

INPUT PARAMETERS:

num_ports — A BYTE value indicating the number of

receivers of a broadcast message. If num_ports = 1, the message is point-to-point.

port_ids — A BYTE vector containing a list of logical unit numbers associated with the intended receivers.

data_buffer — An untyped data buffer containing the contents of the message to be sent.

num_entries — An INTEGER giving the number of data items in the data buffer. NOTE: This may or may not correspond to the number of bytes, depending on the size of each entry.

prec_code — A BYTE indicating the number of bytes required for an entry in the data buffer. Thus prec_code takes on the value of:

1 = byte or character
2 = 2 byte integer or word
4 = long integer or short precision floating point number
8 = long precision floating point number
10 = extended precision or TEMPREAL

OUTPUT PARAMETERS:

message_ids — An array of INTEGERS of sufficient length to hold one message identifier for each entry in the list of receiving ports.

out_status — An INTEGER return code defined to reflect the following conditions:

0 = no exceptional conditions

CALL **start** (num_ids, code_ids, fork_status) (SA only)

INPUT PARAMETERS:

num_ids — A BYTE indicating how many code_ids are to be processed.

code_ids — An INTEGER array indicating which code segments are to be started explicitly by this command. The code identifiers are those previously returned by a call to **send_code**.

OUTPUT PARAMETERS:

fork_status An INTEGER return code defined to reflect the following conditions:

0 = no exceptional conditions

CALL **whoami** (proc_id)

INPUT PARAMETERS: -- NONE --

OUTPUT PARAMETERS:

proc_id An INTEGER returning the physical unit number of the calling processor.

A MINIMAL AND REAL TIME OPERATING SYSTEM FOR MULTIPROCESSING MACHINES

Richard Kefs and David W. Krumme

Massachusetts Institute of Technology,
Tufts University

March 3rd, 1985

1. Introduction

This report describes the design of a minimal, real time, layer-type operating system. It includes the user interface , and implementation on a concurrent machine. It is intented for a wide range of specific applications but its primary purpose is to be efficient, and small. Therefore, no protection mechanism exists, although it can be added easily if one accepts a reduction of the overall performance.

2. General Description

The operating system is composed of six modules which are : memory management ("mem"), process control and scheduling ("pr"), semaphores ("sem"), mailboxes ("box"), naming ("tab"), and time routines ("tm").
There are five layers :

- "mem"
- "pr"
- "sem"
- "tab" and "tm"
- "box".

This means that the operating system can be taylored depending on the application. That is, outer layers can be removed, added or replaced with new ones. This design does not assume common address spaces for processes or processors. But it requires some communication scheme between processors, either by multi-ported sharable memory or communication links. In loosely coupled processor systems where processors have access to sharable and local memory, It is required that each processor must run a self contained collection of the operating system routines. These processors form subsystems which may share and access common ressources such as semaphores, mailboxes, memory pools...
As a result, all sharable objects must be self contained and provide some mechanisms to protect critical sections.
This is achieved when the creation of the object is based upon the processor creator 's knowledge of the environment of that object. When accessed, the object provides information about the type of protection it has - busy waiting, interrupt disabled, serialization, signals...

Consequently, processes may become blocked waiting on some events - ressources not available, others processes in critical sections..- or idle. The major difference between those two states is that the former is synchronously unblocked - as soon as the ressources become available or one process just left the critical section- whereas the latter is asynchronously waked up - as soon as it is noticed . The state of that process is determined by the accessed object. All descriptions in this document are given in C. If a system call fails, a variable "system_status" associated with each process is set with value error, and the return values is unpredictable.

3. Memory Management

The memory allocator works on objects called pools, which are blocks of memory containing a data structure to specify the size of the block, protection mechanisms, type of memory, parent pool from which it can expand... There are two kinds of pools : node pools and memory pools. Node pools are first-fit fast allocator of large numbers of small word-align blocks. Memory pools are used for allocating larger blocks of memory, also in first-fit word-align fashion. The latter does not fragment memory, like node pool, and merges back pieces of memory into larger blocks after being freed. All type of pools can grow by borrowing memory from their parent pool until their maximum size is reached.

Synopsys

```
extern MEM_POOL *mem_create();
extern void mem_delete();
MEM_VAR datasize, maxsize;
MEM_FLAG flag;
MEM_POOL *pool, *parent, mypool;

pool = mem_create(mypool, parent, datasize, maxsize,flag);

mem_delete(pool);
```

Description

Mem_create tries to create a pool at mypool. If mypool is a NULLPOOL, then it creates the pool by allocating memory from the parent pool. The newly created pool has the size datasize. If that pool runs out of memory it will try to borrow memory from the parent pool only if datasize is not greater than the maximum size allowed. The flag specifies the protection -OBJ_CONCURENT, OBJ_SERIALIZE-, the type of the pool - NODE_POOL, MEMORY_POOL-, its nature -MEM_CLEAR, MEM_NONBLOCK-. The flag determines the appropriate

memory allocator routines.

Synopsis

```
extern char *mem_alloc();
extern void mem_dealloc();
MEM_POOL *pool;
char *myblock, *add;
MEM_VAR size;

add = mem_alloc(pool, size);
add = mem_nalloc(pool, size);
add = mem_xalloc(pool, myblock, size);
mem_dealloc(pool, myblock, size);
```

Description

Mem_alloc is the primary allocator routine. It attemps to allocate an area of size bytes from the given pool. If it fails, it changes the state of the calling process according to the pool flag. Mem_nalloc never blocks. Mem_xalloc tries to allocate the space at myblock. Mem_dealloc frees memory space of size size bytes of an area previously allocated from the pool.

4. Process Control

The process control module was design to provide most common operations -blocking, unblocking, priority changes, creation of processes...- in single steps, without any searching, allowing fast scheduling and context switching.

Synopsis

```
extern PCB *pr_create();
extern void pr_exit();
extern void pr_delete();
extern void pr_block();
extern void pr_unblock();
extern void pr_priority();
extern void pr_resched();

PR_PRIORITY priority;
PCB *pcb;
ENV *env;
```

```
pcb = pr_create(env, priority);
pr_exit();
pr_delete(pcb);
pr_block();
pr_unblock(pcb);
pr_piority(pcb, priority);
pr_resched();
```

Description

Pr_create create a pcb for a new process. If memory is available, it sets its environment to env, and its priority to priority. The newly created process is blocked and should be unblock by the calling process in order to be scheduled.
Pr_exit delete the calling process. pr_delete delete a specific process. In actuallity it forces the supposily deleted process to delete itself. Pr_unblock unblocks a process. Pr_priority changes the priority of a specific process replacing that process in the schedule queues and may result in a call to pr_resched. Pr_resched schedule a new process, restoring its environment and saving the environment of the previous running process. The scheduler has prehemptive round-robin priority scheme.

5. Time Keeping

Time is kept by the system in its internal format and seperate routine is provided to encode and decode this internal format.

Synopsis

```
extern void tm_set();
static void tm_inc();
extern TM_VR tm_get();
extern TM_VR tm_encode();
extern void tm_decode();
TM_VR time, interval;
int tickflag;
char buffer[13];
GENERIC (*fnct)(), arg;

tm_set(time);
time = tm_get();
tm_inc();
tm_decode(time, &buffer);
time = tm_encode(buffer);
```

tm_call(interval, tickflag, fnct, arg);

Description

tm_inc is a routine callable at interrupt-time. It updates the time and check for things to be done. Tm_call
sets up a delayed function call and return immediately. After the interval has elapsed the delay call of the form (*fnct)(arg) and will be executed by tm_inc. Typical call to tm would be tm_call(interval, tickflag, unblock, prun) where prun is the calling process, or tm_call(interval, tickflag, sem_notify, sem) where sem is a semaphore. Tm_set sets the time in the internal format. Tm_get return the time int its internal format. Tm_decode converts a time value form its internal format into ascii string of the form yymmddhhmmss, that is 11:59:00 pm Dec 6, 1972 would be converted into 721206235900. Tm_encode performs the reverse operation and returns the time value.

6. Semaphore

```
extern SEM *sem_create();
extern void sem_delete();
extern void sem_wait();
extern void sem_signal();
SEM *semap, *mysemap;
MEM_POOL *pool;
SEM_FLAG flag;
int count;

semap = sem_create(mysemap, pool, flag, count);
sem_delete(semap);
sem_wait(semap);
sem_signal(semap);
```

Description

The sem_create routine creates a semaphore with an initial count of count. If mysemap is not a NULLSEM, then it tries to allocate space via a call to mem_alloc. The flag is used for the protection of critical sections - OBJ_CONCURRENT, OBJ_SERIALIZE. A call to sem_wait decrement and test the count in a undivisible instruction - use of hardware lock otherwise. If the count is negative it acts according to the flag and may result in blocking the calling process. The sem_signal perform the opposite operation and may result in waking up a process.

7. Box

This module is intented for inter process communication and is particularly oriented for uni-processor system with same or different address space or loosely or tightly coupled multi-processor system, but outer layer may be added to support for direct link communication.

Synopsis

```
extern MAIL_BOX *box_create();
extern void box_delete();
extern void box_send();
extern void box_recv();
extern int box_pollsnd();
extern void box_pollrcv();
MEM_POOL *pool;
MAIL_BOX *mbx, *mymbx;
GENERIC *message, *mess;
BOX_FLAG *flag;
int nbmess, sizemess;

mbx = box_create(mymbx, pool,flag, nbmess, sizemess);
box_delete(mbx);
box_send(mbx, message);
box_recv(mbx, mess);
status = box_pollrcv(mbx, mess);
status = box_pollsnd(mbx, message);
```

Description

Box_create create a mailbox. If mymbx is NULLMAIL, it tries to allocate the space from the pool. The mailbox contains nbmess number of messages, each of them of size sizemess bytes. Box_delete performs the reverse operation.

Box_send sends a message. If resources are not available, that is, the receiver mailbox is full, the calling process may be blocked or idle depending on the mailbox flag. Otherwise the message is copied into the mailbox, and the call returns. Box_recv tries to receive a message from a particular process from a specific mailbox. If none is available, the process state depends upon the mailbox flag. Otherwise the message is copied into mess, and is deleted from the mailbox. Box_pollsnd and box_pollrcv operate as box_send and box_recv but do not block the process and return a status of the call.

8. Tab

The purpose of this module is to map logical identities into physical ones. If a process wants to send a message to another process it needs to know the mailbox associated to that process. But since it does not know the id of that process, defined at run time, process must agree at compiled time on unique logical identities. Then they can ask the operating system to associate logical and physical name. As a result all processes can find out physical names from logical names. The overhead is insignificant since one process issues only one call to get the physical identity of a mailbox or a semaphore that it may use heavily.

Synopsis

```
extern TAB_TABLE *tab_create();
extern void tab_delete();
extern GENERIC tab_lookup();
extern void tab_set();
GENERIC hiname, loname;
TAB_TABLE *tab, *mytab;
MEM_POOL *pool;
TAB_FLAG flag;
int nbname, tab_name;


tab = tab_create(mytab, pool, flag, nbname, tabname);
tab = tab_get(tabname);
tab_delete(tab);
tab_set(tab, hiname, loname);
loname = tab_lookup(tab, hiname);
```

Description

Tab_create allocate memory space from the pool for the new table if mytab is NULLTAB. The flag determines the type of protection. Tab_delete deletes the table. Tab_lookup is used to get physical name from logical name.Tab_set will map low name into high name into a specific table. Nbname is the number of names that can be maped. There are a fixed number of different tables : process, semaphore, mailbox, pool. Each of them is determined by tabname. When a new table is created, it is placed in an array of table. Tabname is an index to that array. Therefore every process can retrieve information from a table if and only if it was created and knows its

tabname. Tabname are obviously defined at compiled time. One can issues a call like

```
loname = tab_lookup(tab_get(tabname), hiname);
```

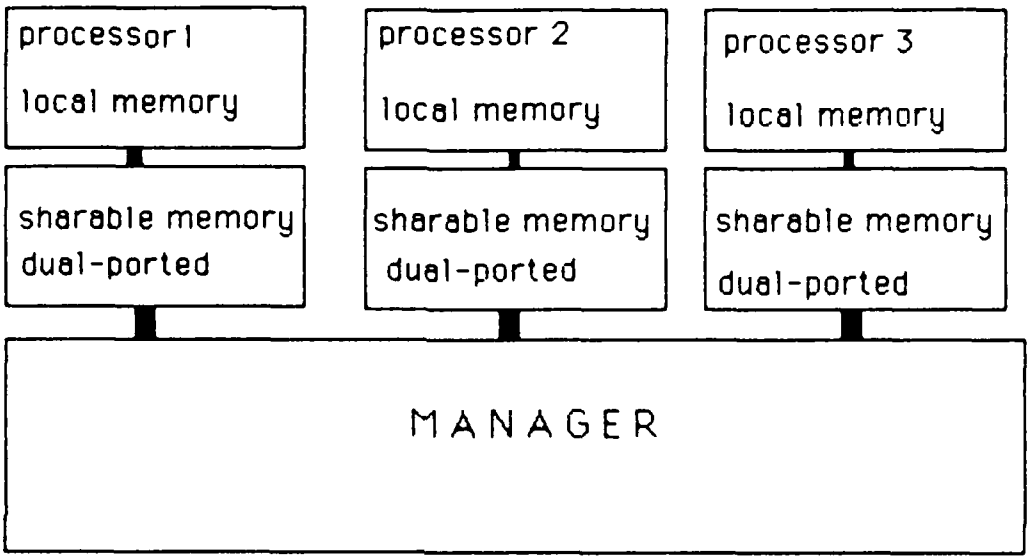
and then

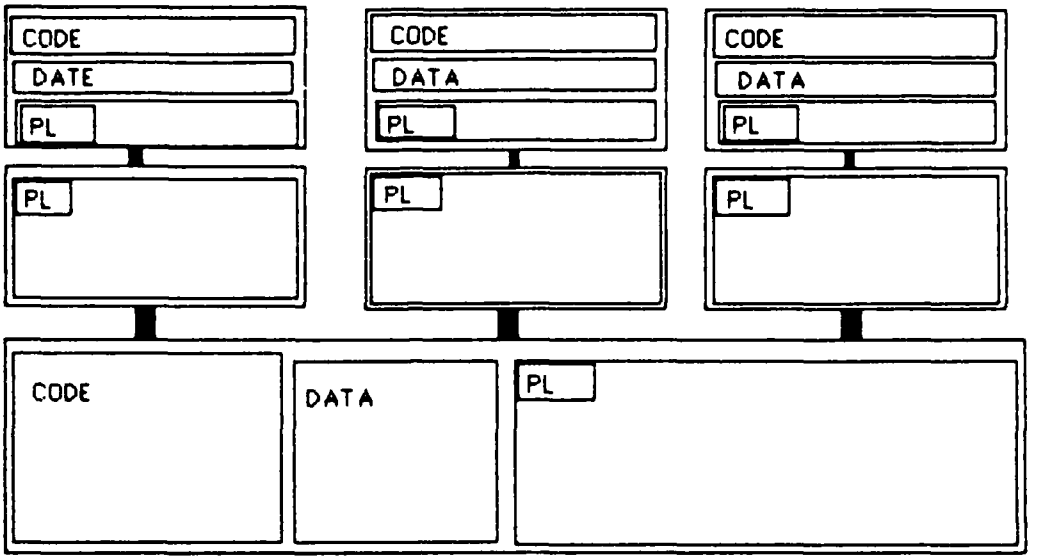
```
box_send(loname, mess);
```

or simply

```
box_send(tab_lookup(tab_get(tabname), hiname), mess);
```

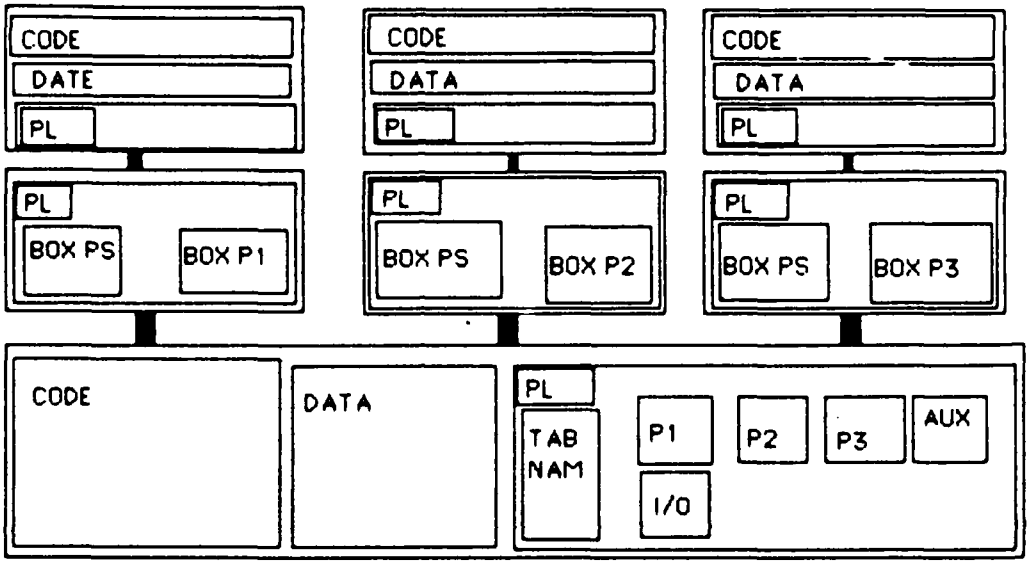
if it will use the mailbox only once.


processor 1
local memory
processor 2
local memory
processor 3
local memory
sharable memory
dual-ported
sharable memory
dual-ported
sharable memory
dual-ported
MANAGER



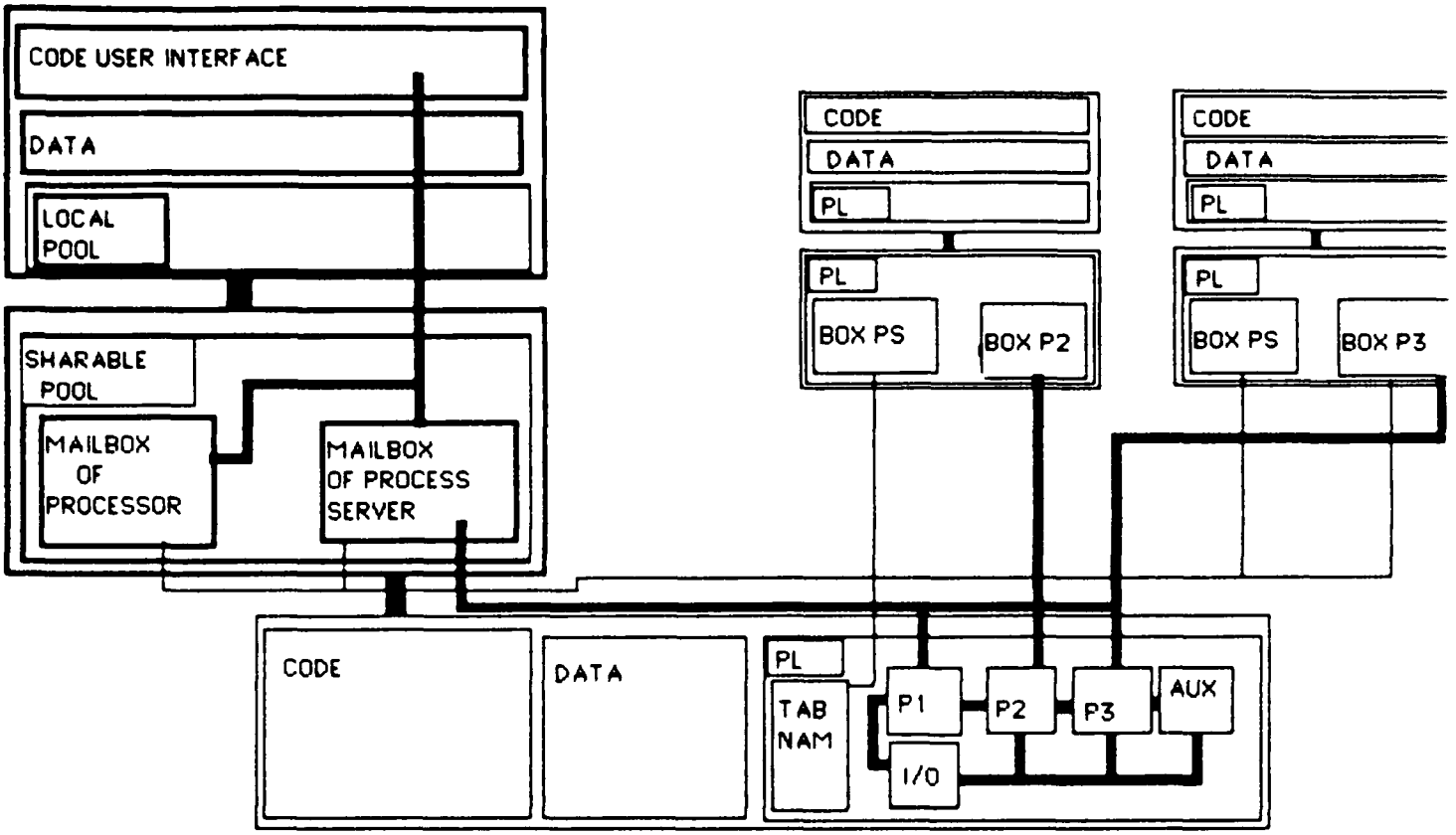
PL : MEMORY POOL
CODE : CODE SEGMENT
DATA : DATA SEGMENT

SYSTEM CONFIGURATION



PL : MEMORY POOL
CODE : CODE SEGMENT
DATA : DATA SEGMENT
PS : PROCESSOR
P* : PROCESS SERVER
BOX : MAILBOX
TAB : TABLE NAME

SYSTEM CONFIGURATION WITH MAILBOX



PL : MEMORY POOL
CODE : CODE SEGMENT
DATA : DATA SEGMENT
PS : PROCESSOR
P* : PROCESS SERVER
BOX : MAILBOX
TAB : TABLE NAME

SYSTEM CONFIGURATION WITH MAILBOX TABLES AND COMMUNICATION PATHS

Fast Toeplitz Orthogonalization Using Inner Products

George CYBENKO

Department of Computer Science, Tufts University, Medford, MA 02155

also of

Statistics Center, Masachusetts Institute of Technology, Cambridge, MA 02139

Abstract

A new method for the orthogonalization of complex m by n Toeplitz matrices is presented. An inverse QR factorization is computed in $10mn + \frac{27}{2}n^2$ multiplications and divisions. This method uses inner products and projections in the same spirit as lattice algorithms for linear prediction do.

1. Introduction

Toeplitz matrices arise in numerous applications of current interest. In a large class of problems, rectangular Toeplitz matrices are specially structured in that they have zeros in the upper right and lower left triangular corners. These matrices arise commonly in linear prediction problems [1, 2] when the *autocorrelation* approach is used. Linear prediction problems, among other problems such as the discretization of integral equations, can also involve rectangular Toeplitz matrices without zero corners (in signal processing contexts these are covariance approaches).

In recent work [3], Sweet described an efficient method for the fast computation of QR decompositions of general Toeplitz matrices. As was noted in that work, there are algorithms in the literature for solving closely related least squares problems involving Toeplitz matrices. In particular, the paper [4] describes an efficient algorithm for solving

---

Research partially supported by contracts AFOSR 82-0210 and NSF MCS-8003364.

the normal equations arising from linear least squares problems involving Toeplitz matrices. Implicit in that work, and related work on *close-to-Toeplitz* matrices, are efficient orthogonalization methods. This paper derives one such fast orthogonalization for complex Toeplitz matrices that uses inner products and so is quite distinct from the method of Sweet.

Our repeated use of the terms *fast* and *efficient* is meant to mean than an order of magnitude less work is involved. In particular, while a general m by n matrix (with $m > n$) can be orthogonalized using mn^2 operations, the fast methods use $O(mn)$ operations.

The method of this paper actually computes an *inverse orthogonal* factorization in the sense that instead of computing

$$T = QR$$

with Q having orthonormal columns and R upper triangular, the method presented here computes a Q and an R for which

$$TR = Q$$

The implication of this for solving least squares problems is quite simple. Instead of backsolving a triangular system, a triangular matrix is multiplied against a vector. In so far as numerical stability is concerned, the computation of the inverse triangular factor presents a potential pitfall. An error analysis of the classical lattice algorithm was presented in [5] and indicated that the excellent numerical properties exhibited in practice by the lattice method can be analytically substantiated. The similarity between the method of this paper and lattice algorithms suggests that some analytical attention ought to be paid to the accuracy of the former. By the same token, the numerical properties of Sweet's method have been subject to study recently [6]. Preliminary investigations suggest that that method is unstable in spite of the use of orthogonal plane rotations (it is the updating and downdating that can present problems there).

In the next section we present a simple derivation of this new orthogonalization method together with a motivation that arises naturally from analogies with lattice algorithms.

2. An Algorithm for Toeplitz Orthogonal Factorization

Let T be a rectangular Toeplitz matrix with entries $t_{i,j} = t_{i-j}$ for simplicity ($1 \le i \le m, 1 \le j \le n, n \le m$). If $t_j = 0$ for $j < 0$ and $m - n < j \le m$, such matrices can be efficiently orthogonalized using the so-called "lattice algorithm" [7, 8, 2]. Since our method for the general case is derived directly from this important special case, it is useful to present this lattice algorithm.

Definition

Z denotes the unit cyclical shift operator -

$$(\mathbf{Z}x)_i = \begin{cases} x_{i-1} & \text{if } 1 < i \leq m \\ x_m & \text{if } i = 1 \end{cases}$$

Note that $T = [x, \mathbf{Z}x, \mathbf{Z}^2x, \ldots, \mathbf{Z}^{n-1}x]$ has a rectangular Toeplitz structure for any m-vector x (in fact, a rectangular circulant structure). We shall assume that $x_j = 0$ for $m-n < j \leq m$ temporarily.

Lattice Algorithm

Input:

$\{t_j\}_{j=1}^{m}$ with $t_j = 0$ for $m-n < j \leq m$

Output:

Orthogonal vectors, q_j with $Q = [q_1, \ldots, q_n]$
and $TR = Q$, R upper triangular.

Initialization:

$$q_1 = x = \hat{q}$$

Main Loop:

FOR $j = 2$ TO n DO

$$k_j = \frac{-(\mathbf{Z}q_{j-1}, \hat{q})}{(\hat{q}, \hat{q})} \tag{1a}$$

$$q_j = \mathbf{Z}q_{j-1} + k_j \hat{q} \tag{2a}$$

$$\hat{q} = k_j \mathbf{Z}q_{j-1} + \hat{q} \tag{3a}$$

Here (x, y) denotes the complex inner product $\sum \bar{x}_i y_i$. See [7, 2, 8] for details.

A key observation is that general Toeplitz matrices can be imbedded into such special Toeplitz matrices of larger dimension $(m+n-2)$ by n as follows.

$$\hat{T} = \begin{bmatrix} t_{-n} & 0 & . & . & 0 \\ t_{-n+1} & t_{-n} & . & . & 0 \\ . & . & . & . & . \\ . & . & . & . & . \\ t_{-1} & t_{-2} & . & . & 0 \\ t_0 & t_{-1} & . & . & t_{-n} \\ t_1 & t_0 & . & . & t_{-n+1} \\ . & . & . & . & . \\ . & . & . & . & . \\ . & . & . & . & . \\ t_{m-1} & t_{m-2} & . & . & t_{m-n} \\ 0 & t_{m-1} & . & . & t_{m-n+1} \\ . & 0 & . & . & . \\ . & . & . & . & . \\ . & . & . & . & . \\ 0 & 0 & 0 & 0 & t_{m-1} \end{bmatrix} = \begin{bmatrix} T_1 \\ T \\ T_2 \end{bmatrix}$$

where T is the original m by n Toeplitz matrix and T_1 and T_2 are lower and upper triangular Toeplitz matrices respectively.

It is convenient to write

$$\hat{T} = [t, \mathbf{Z}t, \mathbf{Z}^2 t, \ldots, \mathbf{Z}^{n-1} t]$$

for the obvious $m+2n-2$ vector t.

Now this $\hat{T}$ has a form suitable for orthogonalization by the previously described lattice algorithm. However, we desire to orthogonalize the columns of $\hat{T}$ with respect to the euclidean inner product over the m interior entries only. We can view this as orthogonalizing the big matrix $\hat{T}$ with respect to a non-Euclidean inner product defined by a weighting say W. This diagonal weighting matrix is an $m+2n-2$ by $m+2n-2$ diagonal matrix

$$W = diag(0, \ldots, 0, 1, \ldots, 1, 0, \ldots, 0)$$

where there are $n-1$ zeros, m ones and $n-1$ zeros respectively. Clearly, if $\hat{T} = \hat{Q}\hat{R}$ is a decomposition of $\hat{T}$ such that $\hat{Q}^* W \hat{Q}$ is diagonal and $\hat{R}$ is upper triangular, then this gives an orthogonal decomposition of T by simply picking off the middle interior m rows of $\hat{Q}$. Our algorithm is based on this observation and really orthogonalizes $\hat{T}$ with respect to the W-inner product.

For notational convenience, let $x^* Wy = [x, y]$. Before proceeding, note the following fundamental identity satisfied by the W-inner product.

$$[\mathbf{Z}x, \mathbf{Z}y] = [x, y] + \bar{x}_{n-1} y_{n-1} - \bar{x}_{n+m-1} y_{n+m-1} \tag{4a}$$

This is easily verified by direct expansion using the definitions. A key element of this identity is that it can be rewritten as

$$[\mathbf{Z}x, \mathbf{Z}y] = [x, y] + x^* e_{n-1} e_{n-1}^* y - x^* e_{n+m-1} e_{n+m-1}^* y = [x, y] + x^* D y$$

where e_{n-1} and e_{n+m-1} are the obvious standard basis vectors and $D = e_{n-1} e_{n-1}^* - e_{n+m-1} e_{n+m-1}^*$ is a rank two "displacement" correction. It will be easy to see shortly that the efficiency of this method depends precisely on the fact that D has small rank, namely two in this case. More general situations where D has rank greater than two but still small will have an order of magnitude more efficient orthogonalization method also.

This algorithm uses one main iteration with two basic loop invariants. They are stated as follows:

Invariant A

For each $j \geq 1$, we have

$$q_j \in \text{span}[t, \mathbf{Z}t, \ldots, \mathbf{Z}^{j-1}t]$$

$$[q_j, q_i] = 0 \ , \ 1 \leq i < j \ , \ q_1 = t$$

Invariant B

For each $j \geq 1$, we have

$$\hat{q}_j \in \text{span}[t, \mathbf{Z}t, \ldots, \mathbf{Z}^{j-1}t]$$

$$[\hat{q}_j, x] = 0 \text{ for all } x \in \text{span}[\mathbf{Z}t, \ldots, \mathbf{Z}^{j-1}t]$$

Given these invariants as true for $1, 2, \ldots, j$ we now demonstrate that they are valid for $j+1$ also. This involves constructing q_{j+1} and $\hat{q}_{j+1}$ from q_j , $i \leq j$ and $\hat{q}_j$, $i \leq j$. By analogy with (2a), let

$$q_{j+1} = \mathbf{Z}q_j + \delta_j \hat{q}_j + \sum_{k=2}^{j} \alpha_{j,k} q_k \tag{5a}$$

First, select δ_j so that

$$[q_1, q_{j+1}] = 0.$$

Thus let

$$\delta_j = \frac{-[q_1, \mathbf{Z}q_j]}{[q_1, \hat{q}_j]} \tag{6a}$$

Now pick i with $2 \leq i \leq j$ and observe that by construction

$$q_i = \sum_{k=1}^{i} \rho_{k,i} \mathbf{Z}^{k-1} t$$

for some $\{\rho_{k,i}\}$. Let

$$\mathbf{Z}p_i = \sum_{k=2}^{i} \rho_{k,i} \mathbf{Z}^{k-1} t$$

$$= \mathbf{Z} \sum_{k=2}^{i} \rho_{k,i} \mathbf{Z}^{k-2} t$$

(that is $\mathbf{Z}p_i$ is the same as q_i but with the single term $\rho_{1,i} q_1 = \rho_{1,i} t$ subtracted). Note that $p_i \in \text{span}[t, \mathbf{Z}t, \ldots, \mathbf{Z}^{i-2}t]$. Using the definitions of δ_j and p_i, we have

$$[q_i, q_{j+1}] = [q_i, \mathbf{Z}q_j] + \delta_j [q_i, \hat{q}_j] + \alpha_{j,i} [q_i, q_i]$$

$$= [\mathbf{Z}p_i + \rho_{1,i} q_1, \mathbf{Z}q_j] + \delta_j [\mathbf{Z}p_i + \rho 1, iq_1, \hat{q}_j] + \alpha_{j,i} [q_i, q_i]$$

$$= [\mathbf{Z}p_i, \mathbf{Z}q_j] + \alpha_{j,i} [q_i, q_i]$$

because both

$$[\rho_{1,i} q_1, \mathbf{Z}q_j] + \delta_j [\rho_{1,i} q_1, \hat{q}_j] = 0$$

(by the previous choice of δ_j) and

$$[\mathbf{Z}p_i, \hat{q}_j] = 0$$

(by Invariant B above). Now, we have

$$[q_i, q_{j+1}] = [\mathbf{Z}p_i, \mathbf{Z}q_j] + \alpha_{j,k} [q_i, q_i]$$

$$= [p_i, q_j] + \overline{p}_{n-1,i} q_{n-1,j} - \overline{p}_{n+m-1,i} q_{n+m-1,j} + \alpha_{j,i} [q_i, q_i]$$

By Invariant A, $[p_i, q_j] = 0$ and so selecting

$$\alpha_{j,i} = \frac{\overline{p}_{n+m-1,i} q_{n+m-1,j} - \overline{p}_{n-1,i} q_{n-1,j}}{[q_i, q_i]}$$

will yield a q_{j+1} with the desired properties. In of itself, this would clearly be and $O(mn^2)$ algorithm and the key to getting an $O(mn)$ algorithm lies in the fact that the summation in (5a) reduces to

$$\sum_{k=2}^{j} \alpha_{j,k} q_k = q_{n+m-1,j} \sum_{k=2}^{j} \overline{p}_{n+m-1,k} \frac{q_k}{[q_k, q_k]} - q_{n-1,j} \sum_{k=2}^{j} \overline{p}_{n-1,k} i \frac{q_k}{[q_k, q_k]}$$

$$= q_{n+\dot{m}-1,j} f_j - q_{n-1,j} b_j$$

Here f_j and b_j are easily updated to give f_{j+1} and b_{j+1} which are required at the next iteration. Specifically, the updating is

$$f_{j+1} = f_j + \overline{p}_{n+m-1,j+1} \frac{q_{j+1}}{[q_{j+1}, q_{j+1}]}$$

$$b_{j+1} = b_j + \overline{p}_{n-1,j+1}\frac{q_{j+1}}{[q_{j+1},q_{j+1}]}$$

Finally, $\hat{q}_{j+1}$ must be obtained and to this end, we note that both $\hat{q}_j$ and q_{j+1} are W-orthogonal to

$$\mathbf{Z}t \ , \mathbf{Z}^2t \ , \ldots , \ \mathbf{Z}^{j-1}t$$

and so any linear combination of these two vectors will also enjoy this orthogonality property. In particular, select γ_j so that

$$\hat{q}_{j+1} = \hat{q}_j + \gamma_j q_{j+1}$$

is W-orthogonal to $\mathbf{Z}^j t$. Thus

$$\gamma_j = \frac{-[\mathbf{Z}^j t, \hat{q}_j]}{[\mathbf{Z}^j t, q_{j+1}]}$$

This outlines the basic algorithm and an actual implementation requires a few additional calculations for bookkeeping purposes. They are:

1. Maintaining ortho*normality* (in the W inner product) of the q_j;
2. Keeping track of R for which $TR = Q$. That is, keeping track of coefficients $\rho_{i,j}$ for which

$$\sum_{j=1}^{i} \rho_{j,i}\mathbf{Z}^{j-1}t = q_i.$$

In light of all this, we have the following formal description of this orthogonalization method. A complete FORTRAN listing is provided in the appendix for double precision Toeplitz matrices. Assume that $t_{i,j} = t_{i-j}$ are the entries of an m by n Toeplitz matrix. The conventions used are:

a. $\mathbf{Z}$ denotes the unit shift operator as described before, with dimension appropriate to the context;

b. **bold roman** letters denote matrices with leading dimensions $n + m + 1$ that are used in the computation of Q in the QR factorization;

c. *italic roman* letters denote scalar quantities, either real or integer;

d. lower case Greek characters ($\beta,\phi,\rho,\hat{\rho}$) denote matrices with leading dimensions n that are used to maintain the coefficients of various important linear combinations (that is, are related to R^{-1} in the QR factorization);

e. as before, [,] denotes the W-inner product.

TOEPLITZ ORTHOGONALIZATION ALGORITHM

:INITIALIZATIONS:

$$k = m + n$$

$$\mathbf{q}_{j,1} = t_{j-n} \quad \text{for } 1 \le j \le k-1 \quad , \quad \mathbf{q}_{k,1} = 0$$
$$\mathbf{f} = \mathbf{b} = \phi = \beta = \rho = 0$$
$$nrm = [\mathbf{q}_1, \mathbf{q}_1]^{1/2}$$
$$\rho_{1,1} = 1/nrm$$
$$\hat{\rho}_1 = 1/nrm$$
$$\mathbf{q}_1 = \mathbf{q}_1/nrm$$
$$\hat{\mathbf{q}} = \mathbf{q}_1$$

:MAIN LOOP:

FOR $j = 2$ TO n DO
BEGIN

$$v = -[\mathbf{Z}\mathbf{q}_{j-1}, \mathbf{q}_1]$$
$$u = -\mathbf{q}_{n-1,j-1}$$
$$w = \mathbf{q}_{n+m-1,j-1}$$
$$\mathbf{q}_j = \mathbf{Z}\mathbf{q}_{j-1} + v\,\hat{\mathbf{q}} + u\,\mathbf{b} + w\,\mathbf{f}$$
$$\rho_j = \mathbf{Z}\rho_{j-1} + v\,\hat{\mathbf{q}} + u\,\beta + w\,\phi$$

$$v = [\mathbf{q}_j, \mathbf{q}_j]$$
$$\mathbf{q}_j = \mathbf{q}_j / v$$
$$\rho_j = \rho_j / v$$
$$v = \mathbf{q}_{n,j} - \rho_{1,j}\, t_0$$
$$w = \mathbf{q}_{n+m,j} - \rho_{1,j}\, t_m$$
$$\mathbf{f} = \mathbf{f} + v\,\mathbf{q}_j$$
$$\phi = \phi + v\,\rho_j$$
$$\mathbf{b} = \mathbf{b} + w\,\mathbf{q}_j$$
$$\beta = \beta + w\,\rho_j$$
$$v = nrm\,\rho_{1,j}$$
$$\hat{\mathbf{q}} = \hat{\mathbf{q}} + v\,\mathbf{q}_j$$
$$\hat{\rho} = \hat{\rho} + v\,\rho_j$$

END

The output of this algorithm contains the orthonormal columns in the vectors consisting of the entries

$$\mathbf{q}_{j,i}$$

as i goes from n to $n+m-1$ inclusively. and the inverse triangular factor in the

matrix ρ.

3. Conclusion

A simple algorithm for computing the inverse orthogonal factorization of a general complex Toeplitz matrix is presented. The method is more efficient for solving linear least squares problems than is the method of Sweet [3]. Our method uses $10mn + \frac{27}{2}n^2$ multiplies while the method in [3] uses $\frac{25}{2}mn$ multiplies. The numerical properties of neither method are well understood yet.

A Fortran program, implementing this method and using calls to LINPACK BLAS [9], can be obtained from the author.

References

1. J. Makhoul, "Linear prediction: a tutorial review," *Proceedings IEEE*, vol. 63, no. 4, pp. 561-580, 1975.
2. G. Cybenko, "A general orthogonalization method with applications to time series analysis and signal processing," *Math. of Comp.*, vol. 40, pp. 323-336, 1983.
3. D.R. Sweet, "Fast Toeplitz orthogonalization," *Numer. Math.*, vol. 43, pp. 1-21, 1984.
4. T. Kailath, S.Y. Kung, and M. Morf, "Displacement ranks of matrices and linear equations," *J. Math. Anal. Appl.*, vol. 68, pp. 395-407, 1979.
5. G. Cybenko, "The numerical stability of lattice algorithms for least squares linear prediction problems," *BIT*, vol. 24, pp. 441-455, 1984.
6. F. Luk, *personal communication*, 1985.
7. F. Itakura and S. Saito, "Digital filtering techniques for speech analysis and synthesis," *Proc. 7th Internat. Congr. Acousti.*, pp. 261-264, Budapest, 1971.
8. J. Makhoul, "A class of all-zero lattice digital filters: properties and applic ations," *IEEE Trans. Acoust. Speech and Signal Process.*, vol. 4, pp. 304-314, 1978.
9. J.R. Bunch et al., *LINPACK User's Guide*, SIAM, Philadelphia, 1979.

Fast Singular Value Decompositions of Some Structured Matrices

George CYBENKO

Department of Computer Science, Tufts University, Medford, MA 02155

also of

Statistics Center, Massachusetts Institute of Technology, Cambridge, MA 02139

Abstract

Recent progress on algorithms for Toeplitz matrices can be advantageously used in the computation of Toeplitz singular value decompositions. This paper presents a method for computing singular value decompositions of m by n Toeplitz matrices that can result in an order of magnitude reduction of work in the case that $m >> n$ which arises often in signal processing applications. The key idea is to preprocess such a matrix by computing its orthogonal decomposition using very fast algorithms. Comparisons with traditional methods are made.

Keywords. singular value decomposition, Toeplitz matrix, orthogonal decomposition.

1. Introduction

A number of recently developed techniques in signal processing make explicit use of the singular value decomposition [1, 2, 3]. This decomposition has been invaluable in a variety of other applications for some time - examples include the regularization of ill-posed problems, statistics, numerical analysis, control and systems theory [4]. The SVD was popularized in the 1960's when stable, efficient, and reliable methods were first

A preliminary version of this work was presented at the International Conference on Acoustics, Speech and Signal Processing, Tampa, Florida, March 1985. Research partially supported by contracts AFOSR 82-0210 and NSF MCS-8003364.

discovered for its computation. Some indication of the SVD's current importance to signal processing is given by the amount of recent research devoted to finding extremely fast, highly parallel algorithms for the computation of general SVD's [5, 6].

The singular value decomposition is easy to describe.

DEFINITION - Given a rectangular m by n complex matrix, A, the *singular value decomposition* of A is a factorization of A of the form

$$A = U\Sigma V^*$$

where $U = [u_1, u_2, \ldots, u_m]$ and $V = [v_1, v_2, \ldots, v_n]$ are m by m and n by n unitary matrices respectively and Σ is an m by n diagonal matrix

$$\Sigma = diag(\sigma_1, \sigma_2, \ldots, \sigma_p)$$

where $p = \min(m, n)$ and $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_p \geq 0$. The columns of U and V are the left and right singular vectors respectively while the σ_j are the singular values of A.

A fundamental fact is that every complex matrix has a singular value decomposition. The reader is referred to standard sources (see [4] for example) for a detailed proof of this and the facts that follow.

The set of singular values $\{\sigma_j\}$ is unique while if $\sigma_i = \sigma_{i+1} = \cdots = \sigma_j$ then the subspaces span$[u_i, \ldots u_j]$ and span$[v_i, \ldots, v_j]$ are unique. Furthermore, if $m > n$ then span$[u_{n+1}, \ldots u_m]$ is unique with a similar statement true for the right singular vectors if $n > m$.

The SVD is evidently closely related to eigendecompositions and, in fact, many of the assertions above can be easily obtained from corresponding facts about eigendecompositions via this relationship.

Specifically, given A, note that $B = A^*A$ is positive semi-definite and Hermitian. Thus B has an eigendecomposition

$$B = V\Lambda V^*$$

with $\Lambda = \text{diag}(\lambda_1, \ldots, \lambda_n)$ and $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n \geq 0$, V unitary. $\Lambda^{1/2}$ is a well defined positive semi-definite diagonal matrix and it is easily verified that

$$(AV\Lambda^{-1/2})^*(AV\Lambda^{-1/2}) = I_n$$

Hence

$$U = AV\Lambda^{-1/2}$$

has orthonormal columns and so $A = U\Lambda^{1/2}V^*$ is the desired singular value

decomposition.

Not only does this relationship show that the singular values of A are essentially the square roots of the eigenvalues of A^*A and that the right singular vectors of A are the eigenvectors of A^*A but it also indicates how the singular value decomposition of A can be computed.

2. Computing the Singular Value Decomposition

Suppose that A is an m by n complex matrix. Note that the SVD of A is simply related to the SVD of PAQ^* where P and Q are unitary. Specifically,

$$PAQ^* = (PU)\Sigma(QV)^*$$

This observation is central to the computation of the SVD since if $B = PAQ^*$ is bidiagonal, then B is extremely sparse and one can then use simple iterations implicitly on B^*B which is Hermitian and tridiagonal to compute its eigenvalues. (A matrix is *bidiagonal* if the only nonzero entries lie along the main diagonal and one of the diagonals adjacent to the main diagonal.) The complexity of reducing an m by n matrix to bidiagonal form is $O(mn^2)$. Depending on how much of the SVD is desired, the computation of the SVD of this bidiagonal matrix can involve work ranging in complexity from $O(m^2)$ to $O(mn^2)$. Clearly, the complexity of bidiagonalizing a matrix is a lower bound on the amount of work required to compute a singular value decomposition. In the case of Toeplitz matrices, it is precisely this step that affords an economy in computation if advantage is made of recent algorithms for Toeplitz matrix orthogonalization. In order to see the role of fast orthogonalization methods, is it important to first take a close look at how the bidiagonalization step can be done.

In general, the bidiagonalization can proceed in two possible ways. Here we are using A to denote an arbitrary m by n matrix.

1. Use a sequence of left *and* right Householder matrices to reduce A to bidiagonal form. This approach was used in the original algorithm by Golub and Reinsch and is currently implemented in LINPACK [7].
2. First compute an orthogonal decomposition of A, say QR, with $Q^*Q = I_n$ and R an n by n upper triangular. Then bidiagonalize R as in 1. above [8].

The advantage of 2. over 1. is that for $m >> n$ there can be as much as a 50% reduction in the amount of work required. This is not obvious but is born out by a careful operation count. In fact, although 2. had been well known in some circles for a few years [9], it was not until an analysis was recently published [8] that method 2. received broad attention.

Thus for general singular value computations, an orthogonalization can roughly half the amount of computation for a large class of important problems. In the next section, we describe algorithms for fast Toeplitz orthogonalization that reduce by an order of magnitude the operation count for this orthogonalization, effecting an even more

substantial computational saving over traditional methods not taking Toeplitz structure into account.

In concluding this section, we extract some entries from a table in [4] that compare the amount of work required by methods 1. and 2. of above. Three distinct problems are listed for SVD computation. They are:

a. computation of Σ, the singular values only;

b. computation of Σ and V;

c. computation of Σ, U_1 and V (here U_1 denotes the first n columns of U).

	Problem	Method 1.	Method 2.
a.	Σ	$2mn^2 - \frac{2}{3}n^3$	$mn^2 + n^3$
b.	Σ , V	$2mn^2 + 4n^3$	$mn^2 + \frac{17}{3}n^3$
c.	Σ , U , V	$7mn^2 + \frac{11}{3}n^3$	$3mn^2 + 10n^3$

Comparisons of Different SVD Methods [4]

3. Algorithms for Toeplitz Orthogonal Factorization

Let T be a rectangular Toeplitz matrix with entries $t_{i,j} = t_{i-j}$ for simplicity ($1 \le i \le m, 1 \le j \le n, n \le m$). If $t_j = 0$ for $j < 0$ and $m - n < j \le m$, such matrices can be efficiently orthogonalized using the so-called "lattice algorithm" [10, 11, 12]. Specifically, the lattice algorithm computes a decomposition of the form $TR = Q$ where Q has orthogonal columns and R is unit upper triangular [12]. Note that in this case, T^*T is a positive-definite Hermitian Toeplitz matrix and can be computed in about $\frac{mn}{2}$ operations. The eigenproblem for T^*T can then be solved in $O(n^3)$ operations. Considering the previously discussed relationship between singular value decompositions and eigenvalue decompositions, this would give an $O(mn + n^3)$ method for problems a. and b. above but $O(mn^2)$ for problem c. Moreover, there are potential accuracy problems with this approach (see [4] for an explanation of this).

The general m by n Toeplitz matrix is really a submatrix of a specially structured Toeplitz matrix as described above. Note that

$$\hat{T} = \begin{bmatrix} t_{-n} & 0 & \cdot & \cdot & 0 \\ t_{-n+1} & t_{-n} & \cdot & \cdot & 0 \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ t_{-1} & t_{-2} & \cdot & \cdot & 0 \\ t_0 & t_{-1} & \cdot & \cdot & t_{-n} \\ t_1 & t_0 & \cdot & \cdot & t_{-n+1} \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ t_{m-1} & t_{m-2} & \cdot & \cdot & t_{m-n} \\ 0 & t_{m-1} & \cdot & \cdot & t_{m-n+1} \\ \cdot & 0 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & 0 & t_{m-1} \end{bmatrix} = \begin{bmatrix} T_1 \\ T \\ T_2 \end{bmatrix}$$

where T is the original m by n Toeplitz matrix and T_1 and T_2 are lower and upper triangular Toeplitz matrices respectively. $\hat{T}$ is a matrix of the type previously described for which the lattice algorithm is suitable.

The lattice algorithm and the Levinson-Durbin algorithm [13] essentially use recursions between sucessive columns of R in an orthogonal decomposition of $\hat{T}$ in order to reduce operation counts. Now, T is a simple submatrix of $\hat{T}$ and the columns of R in an orthogonal decomposition of T also enjoy recurrence relationships albeit much more complex. In a recent paper [14], Sweet interpreted these relations as updating and downdating operations on an orthogonalization. This interpretation was ingeniously exploited by Sweet to obtain an $O(mn)$ algorithm for the orthogonalization of general Toeplitz matrices. Although the algorithm uses orthogonal rotations which themselves are numerically stable building blocks, there are suspicions that Sweet's method may be numerically unsound [15].

Based on a completely different approach, another Toeplitz orthogonalization method based on inner products and projections (just as the lattice algorithm is), has been presented [16].

Sweet's method uses requires about $\frac{25}{2}mn$ operations for the computation of both

Q and R. The method in [16] requires about $10mn + \frac{27}{2}n^2$ operations but computes the *inverse triangular factor*, R where $TR = Q$. This is not an algorithmic obstacle since the SVD of R is obtained from the SVD of R^{-1} by transposition of the matrices of singular vectors and inversion of the singular values. However, the use of the inverse factor raises questions about numerical stability. Although a thorough analysis is not presently available, there are some indications [17] that this family of inner product algorithms for Toeplitz orthogonalization may be numerically well behaved. It would be going too far to say that they are stable for classical QR factorizations but might be stable for *inverse* QR factorizations. An analysis will have to be performed.

The final section identifies the savings that are possible from using this fast Toeplitz orthogonalization as a preprocessing step.

4. Comparisons

The table below should be compared with the previous table using the two general methods currently well known and accepted. This table lists the leading terms of the operation counts for the Toeplitz orthogonalization method presented in [16] together with operation counts for method 1. for the SVD. Note that if $m = n$ then method 1. is more efficient than method 2. Operation counts, as all others, refer to multiplications and divisions.

Problem	Operations
Σ	$10mn + \frac{4}{3}n^3$
Σ , V	$10mn + n^3$
Σ , V , U_1	$10mn + mn^2 + \frac{32}{3}n^3$

Operation Counts for Fast Orthogonalization Preprocessing in SVD

It is evident that orthogonalization as a preprocessing step for Toeplitz SVD computations is advantageous in the cases where $m >> n$. We note that the formation of the cross product matrix, $T^* T$, in all cases has the smallest operation count. In light of possible loss of accuracy in the cross product approach, and the unknown stability

properties of fast Toeplitz orthogonalization, it seems that more work is required before a clear champion emerges. In the meantime, among methods that do not resort to eigenvalue methods for T^*T, it is clear that fast Toeplitz orthogonalization using the method in [16] combined with the classical Golub-Reinsch SVD is fastest for problems where $m >> n$. If accuracy is an absolute requirement, in the absence of thorough anaylses for fast Toeplitz orthogonalization, the QR preprocessing method using Householder tranformations is best (method 2. of above).

References

1. G. Bienvenue and H.F. Hermoz, "New principles of array processing in underwater passive listening," *VLSI and Modern Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1984.
2. D.W. Tufts and R. Kumaresan, "Improved spectral resolution II," *Proceedings of 1980 Int. Conf. Acoust. Speech and Signal Process.*, pp. 592-597, Denver, CO, 1980.
3. R.O. Schmidt, "A Signal Subspace Approach to Multiple Emitter Location and Spectral Estimation," *Ph.D. Thesis*, Stanford University, Department of Electrical Engineering, 1981.
4. G.H. Golub and C.F. Van Loan, *Matrix Computations*, Johns Hopkins, Baltimore, 1983.
5. R. Schreiber, "A systolic architecture for singular value decomposition," *Proc. 1st Internat. Conf. on Vector and Parallel Computing in Scientific Applications*, Paris, 1983.
6. R.P. Brent, F.T. Luk and C. Van Loan, "Computation of the singular value decomposition using mesh-connected processors," *Cornell University, Computer Science Technical Report*, vol. 82-528, 1983.
7. J.R. Bunch et al., *LINPACK User's Guide*, SIAM, Philadelphia, 1979.
8. T. Chan, "An improved algorithm for computing the singular value decomposition," *ACM Trans. on Math. Software*, vol. 8, pp. 72-83, 1982.
9. C.L. Lawson and R.J. Hanson, *Solving Least Squares Problems*, Prentice-Hall, Englewood Cliffs, NJ, 1974.
10. F. Itakura and S. Saito, "Digital filtering techniques for speech analysis and synthesis," *Proc. 7th Internat. Congr. Acousti.*, pp. 261-264, Budapest, 1971.
11. J. Makhoul, "A class of all-zero lattice digital filters: properties and applic ations," *IEEE Trans. Acoust. Speech and Signal Process.*, vol. 4, pp. 304-314, 1978.
12. G. Cybenko, "A general orthogonalization method with applications to time series analysis and signal processing," *Math. of Comp.*, vol. 40, pp. 323-336, 1983.
13. G. Cybenko, "The numerical stability of the Levinson-Durbin algorithm for Toeplitz systems of equations," *SIAM J. Sci. Stat. Comput.*, vol. 1, pp. 303-319, 1980.
14. D.R. Sweet, "Fast Toeplitz orthogonalization," *Numer. Math.*, vol. 43, pp. 1-21, 1984.
15. F. Luk, *personal communication*, 1985.
16. G. Cybenko, "Fast Toeplitz orthogonalization using inner products," *submitted Lin. Alg. and Applications*, 1985.
17. G. Cybenko, "The numerical stability of lattice algorithms for least squares linear prediction problems," *BIT*, vol. 24, pp. 441-455, 1984.

END

FILMED

1-86

DTIC